Center for Applied Autonomous Sensor Systems (AASS)
Örebro University, School of Science and Technology
SE-70182 Örebro
Sweden

# OOP Lab-V: Templates

Generic Programming in C++

In the previous labs and lectures, we have already worked with polymorphism. We worked with overloading (ad-hoc polymorphism) and inheritance (subtyping). Today we start working with templates (parametric polymorphism).

There are three types of templates currently available in C++:

- Variable templates
- Function templates
- Class templates

## *Variable templates*

Variable templates were introduced in C++14, so older compilers would not digest it. Variable templates are useful when you need to define a variable, which exact type can vary. The following example illustrates the situation with a pi number:

```
template <typename T> const T pi = T(3.14159265358979323851);
```

In this example, the value of pi is already known, but the exact type can be specialized later in the code.

To use this variable template, it must be specialized by using the variable name followed by the type in the angle brackets:

```
pi<int>
```

### Exercise 1: Variable Template

Define a pi number in the form of a variable template. Define a function inPiRange(...) which takes an integer argument and returns true if the absolute value of the argument is lower than pi, or false otherwise.

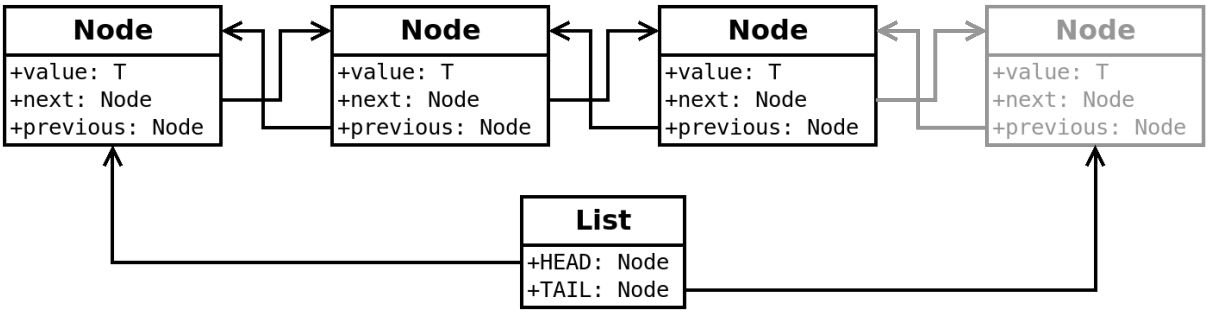
## *Class Templates*

Class templates are probably the most common type of templates used in C++. Majority of STL library (for example, containers) is written using class templates. In this part, you will try to implement one of the containers yourself.

## Exercise 2: Class Template

Doubly-linked list is a container, in which every element has links to its next and previous elements. You need to implement two class templates: one for the element and one for the container (use example from the lecture as a starting point, and a course book). The container class should have the following member functions:

- C-tor
- D-tor
- `begin()`
  - return a pointer to the first element of the list;
- `end()`
  - return a pointer to the last element;




- `front()`
  - returns a reference to the first element;
- `back()`
  - returns a reference to the last element;
- `empty()`
  - returns true if the list if empty;
- `size()`
  - returns current size of the list (integer number);
- `clear()`
  - clear the list;
- **`insert()`**
  - **insert an element at a given position;**
- **`erase()`**


    - **remove an element at a given position;**
- `push_back()`
    - add element to the end;
- `push_front()`
    - add element to the beginning;
- `pop_back()`
    - remove element from the end;
- `pop_front()`
    - remove element from the beginning.

*Tip: it could be smart to start implementation from insert and erase, and then use them in other functions.*

# *Function Templates*

Function templates are very similar to class templates with some limitations. For example, you may not make a partial specialization of function template (we will take a look at it later).

## Exercise 3: Bubble Sort

Bubble sort is the simplest possible sorting algorithm. But it has one more distinct feature: it is the worst performing sorting algorithm. Your task is to implement the function, which will use this algorithm to sort a list. The description of the bubble sort algorithm can be found here: https://en.wikipedia.org/wiki/Bubble_sort

*Tip: the key for bubble sort is swapping neighbouring elements. Depending on how exactly is your list implemented, you might swap the nodes' previous/nest pointers, or swap their content. But please, do not copy the data physically!*